

Universidad ORT Uruguay

OMBÚ

Manual de usuario – Framework para coordinación de sistemas distribuídos, usando el protocolo WS-AtomicTransaction.

María Emilia Blanco, Mónica Hernando, Gabriel Kouyoumdjian, Pablo Venturino
17/02/2010

1 Índice

2 Requisitos

Todas las librerías necesarias para utilizar la API de coordinación están incluidas en el paquete “participant.jar”. Agregar este paquete al classpath de la aplicación es suficiente para poder utilizar la API.

También se necesita una base de datos compatible con el framework Hibernate[1]. La librería de coordinación necesita de un esquema de base de datos dedicado para almacenar información de coordinación.

[1] https://www.hibernate.org/

3 Configuración

Antes de iniciar la aplicación coordinada, debe proveerse cierta información para que la librería de coordinación funcione correctamente. Esta información debe ser provista en un archivo “ombu.properties”, en una subcarpeta “conf” bajo el directorio raíz de la aplicación.

El archivo de configuración “ombu.properties” es un archivo de configuración con pares clave-valor. La siguiente tabla describe las posibles claves y su función en el archivo de configuración.

Clave	Valor
`coordination.service.wsdl`	URL donde está publicado el wsdl del de activación del coordinador. Por ejemplo: `http://localhost:8080/coordinatorWS/ActivationService?wsdl`
`coordination.service.targetNameSpace`	Namespace del servicio que contiene la operación createCoordinationContext, para crear un contexto. Por ejemplo: `http://docs.oasis-open.org/ws-tx/wscoor/2006/06`
`coordination.service.name`	Nombre del servicio que contiene la operación createCoordinationContext. Por ejemplo: `CoordinationService`
`at.services.initiator.binding_url`	URL donde la librería publicará los servicios para el protocolo Completion. Por ejemplo: `http://0.0.0.0:20090/myapp/initiator` (Al usar la dirección 0.0.0.0 el servicio se publica en todas las interfaces activas del equipo).
`at.services.participant.binding_url`	Idem anterior, para los servicios de los protocolos TwoPhaseCommit (la librería los gestiona en un único servicio).
`at.services.initiator.published_url`	URL donde la aplicación coordinadora debería consumir los servicios para el protocolo Completion. Esta URL puede ser distinta a la de la propiedad `binding`. Por ejemplo, si se usa la dirección 0.0.0.0 para publicar los servicios, o si el equipo está detrás de una NAT y debe accederse a través de un servidor virtual.
`at.services.participant.published_url`	Idem anterior, para los servicios de protocolo TwoPhaseCommit.
`persistence_unit`	Nombre de la persistence_unit que se usará para almacenar los datos de coordinación. Más sobre este parámetro a continuación.

3.1 Configuración de la base de datos

La API necesita de una base de datos donde almacenar información de coordinación. El acceso a esta base de datos se realiza a través del framework Hibernate, lo que asegura compatibilidad con casi cualquier base de datos que provea un driver para este framework.

Para configurar el acceso a la base de datos, se necesita:

- Definir la propiedad "`persistence_unit`" en el archivo de configuración de la aplicación (ver siguiente punto).

- Definir una "persistence unit" para la aplicación. Esto debe hacerse en el archivo "persistence.xml" en la carpeta META-INF bajo el directorio raíz de la aplicación. Esta "persistence unit" cumplir los siguientes requisitos.

    - Debe declarar las entidades org.ombu.model.CoordinationContext, org.ombu.model.Participant, org.ombu.model.CompletionParticipant, org.ombu.model.TwoPhaseCommitParticipant.

    - Debe definir la propiedad "`hibernate.hbm2ddl.auto`" con un valor que permita la creación automática de tablas, pero que no sea "`create-drop`", pues se perdería la información de sesiones anteriores y no se podría realizar la recuperación de fallas. Puede usarse, por ejemplo, el valor "`update`".

A continuación se presenta el contenido de un archivo persistence.xml de ejemplo:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="app_pu" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>org.ombu.model.CoordinationContext</class>
    <class>org.ombu.model.Participant</class>
    <class>org.ombu.model.CompletionParticipant</class>
    <class>org.ombu.model.TwoPhaseCommitParticipant</class>
    <properties>
      <property name="hibernate.connection.driver_class"
value="com.mysql.jdbc.Driver" />
      <property name="hibernate.connection.username" value="app_user" />
      <property name="hibernate.connection.password" value="app_passwd" />
      <property name="hibernate.connection.url"
value="jdbc:mysql://dbserver/app_coordination_db " />
      <property name="hibernate.connection.autocommit" value="true" />
         <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5InnoDBDialect" />
      <property name="hibernate.hbm2ddl.auto" value="update"/>
    </properties>
  </persistence-unit>
</persistence>
```

Así mismo, comprobar que la base de datos utilizada tenga soporte para campos de tipo Blob o Text (algunas bases de datos no proveen soporte para este tipo de campos).

Debe asegurarse de que los paquetes con los drivers de Hibernate para acceder a la base de datos estén incluidos en el classpath de la aplicación. Por ejemplo, si se distribuye los drivers de MySQL bajo una carpeta "`lib`" bajo el directorio raíz de la aplicación, puede ejecutarse usando el comando "`java -cp myapp.jar;lib/mysql-connector-java-5.1.7-bin.jar mypackage.MyMainClass`".

4 Ciclo de vida de una transacción

Para iniciar una nueva transacción, se utiliza la clase síngleton `org.ombu.logic.AtomicTransactionManagerImpl`. Esta clase provee los métodos `begin`, `commit` y `rollback` para poder iniciar una transacción y determinar el resultado de ésta.

A continuación se muestra un ejemplo del uso de esta clase.

```
// Obtener la instancia del AtomicTransactionManagerImpl
AtomicTransactionManagerImpl atmgr =
    AtomicTransactionManagerImpl.getInstance();
atmgr.begin();

try {

  // Obtener el contexto. Este se usa para que otros
  // participantes puedan registrarse en esta transacción.
  CoordinationContext ctx = atmgr.getCoordinationContext();

  // Invocar servicios de otros participantes.
  ParticipantA.serviceOne(ctx, business_info);
  ParticipantB.serviceTwo(ctx, business_info);


  // Confirmar la transacción.
  boolean success = atmgr.commit();

} catch (Exception e) {
  // En caso de un error, deshacer la transacción.
  atmgr.rollback();
}
```

La invocación al método ***`begin`*** realiza el llamado al servidor de coordinación para crear una nueva transacción. Internamente, también registra un participante del protocolo Completion para finalmente comunicar el resultado de la transacción.

Los llamados a los métodos de otros participantes deben proveer una forma de compartir el contexto con ellos, de forma que estos también puedan registrarse como participantes en la transacción activa. En el código de ejemplo se asume que las clases `ParticipantA` y `ParticipantB` encapsulan la comunicación con cada uno de los participantes. Esta comunicación no necesita ser a través de servicios web, pero sí debe proveer una forma de compartir el contexto.

Una vez que todas las operaciones de negocio han concluido, se puede invocar al método ***`commit`*** para completar la transacción. La invocación a este método retornará *`true`* si el resultado final de la transacción es exitoso, o *`false`* si por algún motivo la transacción fue abortada (por ejemplo, la falla de un participante o el final del tiempo permitido para la transacción).

En caso de que se decida abortar la transacción (por ejemplo, si uno de los participantes no retorna el resultado esperado) entonces la transacción puede terminarse invocando al método ***`rollback`***, de la misma forma que se invoca el método ***`commit`***.

Es una práctica recomendable que el código de ejecución de la transacción esté dentro de un bloque `try-catch`, y que al levantarse un error se invoque al método ***`rollback`***. Esto logra que ante cualquier evento inesperado durante la ejecución aborte la transacción.

5 Publicación de servicios coordinados

Los servicios que una aplicación de negocios exponga a otras aplicaciones no necesariamente deben ser publicados como servicios web. Sin embargo, la coordinación de las aplicaciones sí se llevará a cabo mediante servicios web, por lo que el primer paso para lograr una aplicación coordinada es publicar los servicios necesarios para la coordinación. Esto se hace invocando, en la clase *AtomicTransactionManagerImpl*, al método *initializeInstance*, o en su defecto, al método *getInstance* (que a su vez invoca al primero al ser llamado por primera vez).

```
public static void main(String[] args) {

    // Al inicio de la aplicación, publicar los servicios de coordinación.
    AtomicTransactionManagerImpl.initializeInstance();

    // Resto del código de la aplicación.
    ...
}
```

Para crear un servicio que pueda ser coordinado, deben realizarse varios pasos. Primero, se debe recibir el contexto de la transacción en la que se va a participar. Luego se debe crear una clase que implemente la lógica de negocio y registrarla para que se ejecute al recibir mensajes de coordinación para un participante con esta operación. Por último, al momento de ejecutar una operación de negocio, se debe registrar un participante en la transacción utilizando el contexto recibido y la operación a la que el participante corresponde. A continuación se brindan más detalles de este proceso.

5.1 Recibir el contexto de la transacción

El servicio en cuestión debe recibir como parámetro un objeto *CoordinationContext*, o en su defecto, proveer alguna forma de recibir este objeto.

5.2 Crear un Manager para la operación

Para cada servicio de negocio que la aplicación publique, se deberá crear una nueva clase que contenga la lógica de coordinación de este servicio. Estas clases serán las encargadas de procesar los mensajes de coordinación de cada participante que pertenezca a los servicios que éstas clases manejan.

Estas clases *manager* deberán heredar de la clase abstracta *org.ombu.participant.at.TwoPhaseCommitParticipantManager*, e implementar los métodos *onPrepare*, *onCommit* y *onRollback*. A continuación se muestra un ejemplo de esta clase.

```java
static class DebitManager extends TwoPhaseCommitParticipantManager {

  @Override
  public Vote onPrepare(TwoPhaseCommitParticipant participant) {
    // Implementar la lógica de preparación.
    return Vote.PREPARED;
  }

  @Override
  public void onCommit(TwoPhaseCommitParticipant participant) {
    // Implementar la lógica de confirmación.
  }

  @Override
  public void onRollback(TwoPhaseCommitParticipant participant) {
    // Implementar la lógica de cancelación.
  }
}
```

La clase creada implementará estos tres métodos, que corresponden con cada uno de los pasos en una transacción atómica del proceso de confirmación en dos fases (Two Phase Commit). Como se muestra, en cada uno de los métodos se tiene acceso, como parámetro, a un objeto del tipo *TwoPhaseCommitParticipant*. Con este objeto se puede acceder al identificador único del participante con un llamado a `participant.getId()`, o el estado del participante con un llamado a `participant.getState()`.

onPrepare

En este método se debe implementar la lógica de preparación del servicio. Este método es invocado por el coordinador en primera instancia para determinar si todos los participantes pueden confirmar o cancelar la operación sin repercusiones. Por lo tanto, la implementación de este método debe dejar al sistema en un estado en el que, tanto si el siguiente mensaje recibido es `commit`, o si es `rollback`, se asegura que las dos operaciones podrán realizarse sin errores.

Este método retorna un objeto de tipo *Vote*. Este objeto determina el resultado del método, y el sistema lo utiliza para saber si la transacción puede continuar y cómo manejar a este participante durante el resto de la coordinación. Los valores que presenta este enumerado son los siguientes:

Valor	Significado
`Vote.PREPARED`	La operación de preparación ha finalizado con éxito. Retorne este valor si puede asegurar que, en el estado actual del sistema, puede ejecutar con éxito tanto la operación `onCommit` como la operación `onRollback`.
`Vote.ABORTED`	La operación de preparación no pudo finalizar con éxito. Retorne este valor si surge algún error durante la ejecución de la preparación, o si no puede asegurar que tanto la operación `onCommit` como la

	operación `onRollback` pueden terminar con éxito. Si retorna este valor, asegúrese de cancelar la operación (ya sea ejecutando la lógica del método `onRollback` u otra lógica acorde con el estado del sistema. Una vez el coordinador reciba un mensaje de `Aborted` de este participante, no intentará enviar el mensaje de `Rollback` al determinar el resultado de la transacción.
`Vote.`***`READONLY`***	La operación de preparación tuvo éxito, y además, la posterior ejecución de `onCommit` o de `onRollback` llegarían al mismo resultado. Retorne este valor si se da el caso en que, sin importar la resolución de la transacción, el estado de la aplicación resultaría idéntico. Si retorna este valor, asegúrese de ejecutar el resto de la lógica de negocio (si es que existe y corresponde hacerlo) para alcanzar el resultado final de la transacción. Una vez el coordinador reciba un mensaje de `ReadOnly` de este participante, no intentará enviar los mensajes de `Commit` o `Rollback` al determinar el resultado de la transacción.

Si la ejecución del método *`onPrepare`* retorna un valor *null*, se asumirá `Vote.`*`ABORTED`* como resultado de la operación.

onCommit

En este método se debe implementar la lógica a realizar al recibir el mensaje de `commit`. Usualmente, el resultado de ejecutar este método es hacer permanentes los cambios realizados en el método *`onPrepare`*.

onRollback

En este método se debe implementar la lógica a realizar al recibir el mensaje de `rollback` desde el coordinador. Un mensaje de `rollback` significa que el coordinador determinó que la transacción no se puede confirmar con éxito, por lo que todos los participantes deben deshacer sus cambios.

La lógica de este método debe entonces revertir los cambios en el sistema hechos durante la ejecución de la lógica de *`onPrepare`*. Si bien se puede optar por dejar información de auditoría o indicios de que la transacción se intentó llevar a cabo, el resultado final no debe tener ningún impacto en el resto del sistema.

5.2.1 Estrategias para la implementación

Existen varias alternativas sobre cómo implementar los métodos *onPrepare*, *onCommit* y *onRollback* de forma que el resultado final sea consistente. La siguiente tabla describe algunos de los posibles enfoques.

onPrepare	**onCommit**	**onRollback**	**Comentarios**
Almacenar los cambios planificados sin ejecutarlos.	Ejecutar los cambios almacenados.	Borrar los cambios almacenados, sin haberlos ejecutado.	La ejecución de `onPrepare` podría incluir validaciones.
Ejecutar los cambios, haciéndolos visibles. Guardar información para deshacerlos.	Eliminar la información para deshacer los cambios.	Ejecutar las acciones para deshacer los cambios.	Es un enfoque de compensación.
Almacenar el estado original, prevenir acceso externo, ejecutar los cambios.	Permitir el acceso (liberar los bloqueos).	Restaurar el estado original, permitir el acceso (liberar los bloqueos).	Este es un enfoque típico en una base de datos.
Ejecutar los cambios, marcarlos como provisorios, hacerlos visibles.	Marcar los cambios realizados como definitivos.	Eliminar los cambios realizados, o marcarlos como cancelados.	Este enfoque conserva evidencia del intento de transacción.

5.3 Registrar el Manager para los servicios

Contando ahora con la clase Manager, es necesario asociar esta clase a una operación, para poder seleccionarla en el momento de recibir los mensajes de `prepare`, `commit` o `rollback`. Dado que la API publica un único servicio para escuchar estos mensajes de coordinación, pero la aplicación provee varios servicios coordinados, es necesario diferenciar al momento de recibir uno de estos mensajes cuál es el servicio al que pertenece.

Para poder diferenciar y seleccionar la lógica de negocio que se debe ejecutar, según el participante destinatario de los mensajes de coordinación, es necesario registrar en el servicio de coordinación una instancia de esta clase, y asociarle una palabra clave que la diferencie de los demás Managers. A esta palabra clave la llamamos "*operación*".

Esta operación es la que diferencia a cada Manager de los demás, de forma de poder seleccionar la lógica de negocio a ejecutar. Al momento de registrar un participante en la transacción (esto se verá más adelante) se le asociará también una operación, de modo que pueda determinarse el Manager que le corresponde.

Para registrar un Manager, se utiliza la clase *TwoPhaseCommitParticipantService*. Esta clase es la encargada de ejecutar toda lógica de coordinación para los participantes de TwoPhaseCommit, y sólo necesita de una instancia de un Manager para ejecutar la lógica correspondiente al negocio.

El siguiente código muestra cómo registrar un Manager para poder ejecutar su lógica.

```
public static void main(String[] args) {

  // Inicializar los servicios del coordinador.
  AtomicTransactionManagerImpl.initializeInstance();

  // Registrar una instancia de DebitManager para ejecutar
  // la lógica de la operación "debit".
  TwoPhaseCommitParticipantSevice.getInstance()
      .registerParticipantManager("debit", new DebitManager());

  // Publicar el servicio de negocio como un servicio web.
  String debitUrl = "http://localhost:8081/BankApplication";
  Endpoint.publish(debitUrl, new BankApplication());

}
```

5.4 Registrar participantes en la transacción

Una vez que se implementó la lógica de negocio en las clases Manager, se debe conseguir que esta lógica sea ejecutada. Además, la lógica que se ejecutará en la invocación a los servicios de negocio no ha sido definida. Es en esta lógica que se lleva a cabo el registro de un participante en la transacción.

Durante la ejecución de un método de un servicio de negocio coordinado, no se ejecutará la lógica de negocio como se haría en un servicio no coordinado. En su lugar, la invocación al servicio de negocio debe cumplir dos funciones:

- Primero, registrar un nuevo participante en la transacción.
- Segundo, almacenar la información de negocio intercambiada (los parámetros recibidos) para poder acceder a ella desde los métodos del Manager.

El registro del nuevo participante se realiza a través del método *registerTwoPhaseCommitParticipant*, que provee la clase *AtomicTransactionManagerImpl*, Este método registra un nuevo participante en la transacción (invocando al servicio correspondiente del coordinador) y retorna un objeto del tipo *TwoPhaseCommitParticipant*. Este método recibe los siguientes parámetros.

Parámetro	Descripción
`CoordinationContext context`	Es el contexto obtenido de la transacción en la que se quiere registrar el participante. Este objeto probablemente haya sido enviado desde la aplicación que inició la transacción, a través de los parámetros del servicio de negocio.
`String operation`	El nombre de la operación de negocio. Este nombre se utilizará al recibir mensajes de coordinación para determinar cuál es el Manager que se debe ejecutar.
`TwoPhaseCommitParticipant.Type type`	Opcional. Determina el tipo de protocolo TwoPhaseCommit que se utilizará: `Durable` o `Volatile`. Por defecto: `Durable`.

[todo: Log]

[todo: distribuir una aplicación de ejemplo]